ANÁLISIS Y PRUEBAS DE DESEMPEÑO EN DISPOSITIVOS AP MEDIANTE IMPLEMENTACIÓN DE UN FIRMWARE BASADO EN EL KERNEL DE LINUX
ANEXO

Universidad
del Cauca

ALBERTO FERNANDO RODRÍGUEZ PABÓN
WILLIAM LEANDRO MORENO CASTRO

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Telecomunicaciones
Popayán
2007

ANÁLISIS Y PRUEBAS DE DESEMPEÑO EN DISPOSITIVOS AP MEDIANTE IMPLEMENTACIÓN DE UN FIRMWARE BASADO EN EL KERNEL DE LINUX
ANEXO

ALBERTO FERNANDO RODRÍGUEZ PABÓN
WILLIAM LEANDRO MORENO CASTRO

Universidad
del Cauca

Trabajo de Grado para optar al título de
Ingeniero en Electrónica y Telecomunicaciones

Director:
Ing. Esp. Guefry Agredo Méndez

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Telecomunicaciones
Popayán
2007

TABLA DE CONTENIDO

pag.
1.1 *Java Desktop System*....1
2 EL COMPILADOR GCC.... 3
2.1 Perspectiva general....3
2.2 Lenguajes Soportados....3
2.3 Arquitecturas Soportadas....4
2.4 Estructura....4
2.5 `Front Ends`....5
2.6 `Back end`....5
2.7 Sintaxis....5
2.8 Ejemplos.....5
2.9 Sufijos en nombres de archivo.....6
2.10 Opciones. 6
2.11 Etapas de compilación. 7
2.12 Preprocesado. 7
2.13 Compilación.....8
2.14 Ensamblado. 8
2.15 Enlazado....8
2.16 Todo en un solo paso.....9
3. DESCRIPCIÓN DE LIBRERÍAS PARA LA COMPILACIÓN DEL FIRMWARE.... 10
3.2 `Binutils`....10
3.2.1 Descripciones cortas....10
3.2.2 Dependencias de instalación de `Binutils`....11
3.3 `Patch`....11
3.3.1 Descripción corta....11
3.3.2 Dependencias de instalación de `Patch`....11
3.4 `Bzip2`....12
3.4.1 Descripciones cortas....12
3.4.2 Dependencias de instalación de `Bzip2`....12
3.5 `Flex`....12
3.5.1 Descripciones cortas....13
3.5.2 Dependencias de instalación de `Flex`....13
3.6 `Bison`....13
3.6.1 Descripciones cortas....13
3.6.2 Dependencias de instalación de `Bison`....13
3.7 `Make`....14
3.7.1 Descripción corta....14
3.7.2 Dependencias de instalación de `Make`....14
3.8 `Gettext`....14
3.8.1 Descripciones cortas....14

3.8.2 Dependencias de instalación de Gettext 16
3.9 Glibc 16
3.9.1 Descripciones cortas 16
3.9.2 Dependencias de instalación de Glibc 18
4 ENLACE DINÁMICO Y ESTÁTICO DE LIBREIRAS 19
4.1 Enlace Estático 19
4.2 Enlace Dinámico 19
5 UTILIDAD MAKE 21
5.1 Variables de entorno 22
5.2 El archivo Makefile 22
5.3 Mejora del Makefile 23
5.4 Variables en el Makefile 25
5.5 Varios makefiles 26

1 SUSE LINUX

SUSE Linux es una de las más conocidas distribuciones Linux existentes a nivel mundial. Entre las principales virtudes de esta distribución se encuentra el que sea una de las más sencillas de instalar y administrar, ya que cuenta con varios asistentes gráficos para completar diversas tareas.

Su nombre "SUSE" es el acrónimo del alemán "*Software- und Systementwicklung*", el cual formaba parte del nombre original de la compañía y que se podría traducir como "desarrollo de software y sistemas". El nombre actual de la compañía es SUSE Linux, habiendo perdido el primer término de su significado.

La adquisición por la compañía multinacional estadounidense Novell se llevó a cabo en enero de 2004. En el año 2005, siguiendo los pasos de RedHat Inc., anunció la liberación de la distribución SUSE Linux, para que sea la comunidad la encargada del desarrollo de esta distribución, que ahora se denomina *openSUSE*.

SUSE incluye un programa único de instalación y administración llamado *YaST2* que permite realizar actualizaciones, configurar la red y el *firewall*, administrar a los usuarios, y muchas más opciones todas ellas integradas en una sola interfaz. Además incluye varios escritorios, entre ellos los más conocidos que son *GNOME (GNU Network Object Model Environment)* y *KDE* (*K Desktop Environment)*, siendo este último el escritorio por defecto. La distribución incorpora las herramientas necesarias para redistribuir el espacio del disco duro permitiendo así la coexistencia con otros sistemas operativos existentes en el mismo.

SUSE utiliza sistemas de paquetes de *RedHat* (*RPM, RedHat package manager*) aunque no guarda relación con esta distribución, esta basada en *Slackware*.

Distribuciones Linux basadas en SUSE Linux

- *Novell Linux Desktop*
- *Java Desktop System*
- *SUSE Linux Enterprise Edition*

1.1 *Java Desktop System*

Java Desktop System, es el nombre final que recibió el proyecto *Mad Hatter*, que vio la luz a finales del 2003, el objetivo se centra en el desarrollo de un Linux amigable acompañado de su paquete *Star Office* y *Java*, además de *Gnome, Mozilla, Evolution ó Gaim* y está basado en estándares abiertos.

En un principio *Java Desktop System* se basaba en la distribución SUSE, que fue comprada a posteriori por Novell, pero más tarde los ejecutivos de Sun decidieron que *Java Desktop System* se base en *Solaris*, su otro sistema operativo. Así los planes de *Sun* incluyen de esta manera unificar la interfaz de usuario de sus sistemas basados en Linux con los basados en *Solaris*. Para algunos críticos sólo es un intento por ganar algunos usuarios para *Solaris* y, a largo plazo, ofrecer una versión unificada en el escritorio de equipos personales y estaciones de trabajo.

Java Desktop System ha conseguido cierto éxito en mercados asiáticos. Según los acuerdos, Sun proporciona un millón de unidades del *Java Desktop System* al año, acuerdo que proveerá de un

sistema operativo a China a la espera de que fructifique su acuerdo con Japón y Korea para el desarrollo de un sistema propio, tendencia clara en los chinos, que están elaborando su propia alternativa al DVD, los llamados EVDs.

Java Desktop System actualmente está en su versión 3, en esta nueva versión sigue manteniendo el software con el que *Sun* quiere competir con Windows: escritorio *Gnome 2.2*, suite de navegación *Mozilla*, ofimática con *StarOffice*, *Gaim*, *Real Player* y *Ximian Evolution*.

Sun Java Desktop System, es la primera alternativa viable a *Microsoft Windows*. *Java Desktop System* es un sistema operativo económico y seguro diseñado para expandirse en el actual mundo dominado por sistemas con *Windows*. Es el único entorno con total integración con la tecnología *Java*, haciendo posible ejecutar sin ninguna modificación miles de aplicaciones *Java* bajo el mismo aspecto gráfico.

1.2 Características básicas

- Económico: Menor coste de compra frente al sistema operativo *Microsoft Windows* y su suite ofimática *Microsoft Office XP*.

- Menor Complejidad: Totalmente integrado en un solo paquete para facilitar la instalación, gestión y manejo.

- Seguridad: El estricto control de acceso a `root` en Linux/*Unix* previene virus que modifican ficheros. *Java*, con su infraestructura de seguridad controla el entorno del sistema ante posibles ataques de virus.

- Interoperabilidad: Ficheros y documentos pueden ser compartidos e impresos en entornos *Windows* y *Unix*; Los sistemas existentes se mantienen sin alteración, incorpora soporte para nuevos idiomas como coreano, japonés y portugués/brasileño.

1.3 Empresas que usan SUSE Linux

- *Novell*
- *Sun Microsystems*
- *IBM*
- *Fujitsu*

2 EL COMPILADOR GCC

GNU Compiler Collection es un conjunto de compiladores creados por el proyecto *GNU. GCC* es software libre y lo distribuye la fundación para el software libre (*FSF, Free Software Foundation*) bajo la licencia *GPL*. Estos compiladores se consideran estándar para los sistemas operativos similares a *UNIX* de código abierto y también para algunos sistemas operativos propietarios derivados de ellos, como *Mac OS X*. *GCC* requiere el conjunto de aplicaciones conocido como `binutils` para realizar tareas como identificar archivos objeto u obtener su tamaño para copiarlos, traducirlos, crear listas, enlazarlos, o quitarles símbolos innecesarios.

Originalmente *GCC* significaba *GNU C Compiler* (compilador *GNU* para *C*), porque sólo compilaba el lenguaje C. Posteriormente se extendió para soportar *C++, Fortran, Ada* y otros.

2.1 Perspectiva general

GCC fue escrito originalmente por Richard Stallman en 1987 como el compilador para el proyecto *GNU*. Su desarrollo fue guiado por la *FSF*.

En 1997, un grupo de desarrolladores, insatisfechos con la lentitud del avance del proyecto y sabiendo que el código en el que se estaba trabajando estaba restringido a un pequeño grupo de programadores iniciaron el proyecto sistema de compiladores experimentales/mejorados *GNU* (*EGCS, Experimental/Enhanced GNU Compiler System*,), el cual mezcló diferentes proyectos. El desarrollo de *EGCS* probó tener más vida que el de *GCC* y fue finalmente escogido como la versión oficial de *GCC* en abril de 1999.

GCC ahora es mantenido por un variado grupo de programadores alrededor del mundo. Estos compiladores soportan más microprocesadores y sistemas operativos que ningún otro.

GCC ha sido adoptado para compilar y desarrollar un gran número de sistemas, incluyendo *GNU/Linux, BSD, Mac OS X, NeXTSTEP y BeOS.*

GCC es frecuentemente el compilador elegido para desarrollar software que debe correr sobre diferentes arquitecturas. Las diferencias entre compiladores nativos llevan a dificultades en el desarrollo, al usar *GCC*, el mismo compilador es usado en todas las plataformas. El código resultante puede ser ligeramente más lento, pero la potencial reducción de los costos del desarrollo hace que valga la pena utilizarlo.

2.2 Lenguajes Soportados

En un compilador, el `front end` traslada el lenguaje del código fuente a una representación intermedia que a su vez funciona con el `back end` para producir en la salida el código. En la versión 4.0.0 (liberada el 20 de abril de 2005), se incluyen `front ends` para:

- *Ada (GNAT)*
- *C (GCC)*
- *C++ (G++)*
- *Fortran (GFortran)*

- *Java (GCJ)*
- *Objective C*

Anteriormente se incluía un `front end` para *CHILL*, pero fue desechado debido a la falta de mantenimiento. El `front end` G77 fue abandonado en favor del nuevo *GFortran* que soporta *Fortran 95*. También existen `front ends` para *Pascal*, *Modula-2, Modula-3, Mercury, VHDL y PL/I*, pero no se incluyen en la distribución principal.

2.3 Arquitecturas Soportadas

En la versión 3.2 se incluye soporte para:

- *Alpha*
- *ARM*
- *H8/300*
- *System/370, System 390*
- *x86 and x86-64*
- *IA-64 "Itanium"*
- *Motorola 68000*
- *Motorola 88000*
- *MIPS*
- *PA-RISC*
- *PDP-11*
- *PowerPC*
- *SuperH*
- *SPARC*
- *VAX*

También hay soporte para arquitecturas menos conocidas: *A29K, ARC, Atmel AVR, C4x, CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, NS32K, ROMP, Stormy16, V850, y Xtensa.* Se ha dado soporte para procesadores adicionales, como *D10V, PDP-10, y Z8000* en versiones mantenidas separadas de la versión de la *FSF*.

2.4 Estructura

La interfaz exterior de *GCC* es generalmente estándar para un sistema *UNIX*. Los usuarios llaman un programa controlador llamado *GCC*, que interpreta los argumentos dados, decide que compilador usar para cada archivo y ejecuta el ensamblador con el código resultante, después ejecuta el enlazador para producir un programa completo.

Cada uno de los compiladores es un programa independiente que toma como entrada código fuente y produce código en ensamblador. Todos ellos tienen una estructura interna común: un `front end` por lenguaje que procesa el lenguaje y produce un árbol de sintaxis y un `back end`, que convierte esos árboles al lenguaje de transferencia de registros *(RTL, Register Transfer Language*) de *GCC*, luego realiza varias optimizaciones y produce el código en ensamblador utilizando un reconocimiento de patrones específico para la arquitectura, originalmente basado en un algoritmo de Jack Davidson y Chris Fraser.

Casi todo *GCC* está escrito en *C*, aunque gran parte del `front end` de Ada está escrito en Ada.

2.5 Front Ends

Los front ends varían internamente, teniendo que producir árboles que puedan ser manejados por el back end. Todos los analizadores son recursivos descendentes y fueron escritos manualmente, no generados automáticamente.

Hasta hace poco, el árbol de representación de programa no era totalmente independiente del procesador para el que se quería general el código. Recientemente se han incluido dos nuevas formas de árbol independientes del lenguaje. Estos nuevos formatos son llamados *GENERIC* y *GIMPLE*. El análisis ahora es realizado creando árboles temporales dependientes del lenguaje y convirtiéndolos a *GENERIC*. El *gimplifier* convierte esto a *GIMPLE*, que es el lenguaje común para un gran número de optimizaciones independientes de la arquitectura y del lenguaje.

La optimización en árboles no entra en lo que la mayoría de los desarrolladores de compiladores consideran trabajo del front end, ya que no es dependiente del lenguaje y no involucra el análisis. Los desarrolladores de *GCC* han dado a esta parte del compilador el nombre de middle end. Las optimizaciones incluyen eliminación de código que nunca se ejecuta, eliminación parcial de redundancia, redundancia a la hora de evaluar expresiones. Actualmente se está trabajando en optimizaciones basadas en dependencia de arreglos.

2.6 Back end

El comportamiento del back end está parcialmente especificado por el preprocesador de macros específicas a la arquitectura objetivo, por ejemplo para definir la posición de los bits más significativos, tamaño de palabra, convención para llamadas, etc. El back end utiliza éstas para la generación de *RTL*, aunque en *GCC* éste es independiente del procesador, la secuencia inicial de instrucciones abstractas es adaptada a la arquitectura objetivo.

2.7 Sintaxis

```
gcc [ opción | archivo ] ...
g++ [ opción | archivo ] ...
```

Las opciones van precedidas de un guión, como es habitual en *UNIX*, pero las opciones en sí pueden tener varias letras; no pueden agruparse varias opciones tras un mismo guión. Algunas opciones requieren después un nombre de archivo o directorio, otras no. También pueden darse varios nombres de archivo a incluir en el proceso de compilación.

2.8 Ejemplos.

```
gcc hola.c
```

Compila el programa en *C* hola.c, genera un archivo ejecutable a.out.

```
gcc -o hola hola.c
```

Compila el programa en *C* hola.c, genera un archivo ejecutable hola.

```
g++ -o hola hola.cpp
```

Compila el programa en *C++* hola.c, genera un archivo ejecutable hola.

```
gcc -c hola.c
```

No genera el ejecutable, sino el código objeto, en el archivo `hola.o`. Si pregunta un nombre para el archivo objeto, se usa el nombre del archivo en *C* y le cambia la extensión por `.o`.

```
gcc -c -o objeto.o hola.c
```

Genera el código objeto indicando el nombre de archivo.

```
g++ -c hola.cpp
```

Igual para un programa en *C++.*

```
g++ -o ~/bin/hola hola.cpp
```

Genera el ejecutable `hola` en el subdirectorio `bin` del directorio propio del usuario.

```
g++ -L/lib -L/usr/lib hola.cpp
```

Indica dos directorios donde el compilador buscará las bibliotecas. La opción *-L* debe repetirse para cada directorio de búsqueda de bibliotecas.

```
g++ -I/usr/include hola.cpp
```

Indica un directorio para buscar archivos de encabezado (de extensión `.h`).

2.9 Sufijos en nombres de archivo.

Son habituales las siguientes extensiones o sufijos de los nombres de archivo:

`.c`	fuente en C
`.C .cc .cpp .c++ .cp .cxx`	fuente en *C++;* se recomienda *.cpp*
`.m`	fuente en *Objective-C*
`.i`	*C* preprocesado
`.ii`	*C++* preprocesado
`.s`	fuente en lenguaje ensamblador
`.o`	código objeto
`.h`	archivo para preprocesador (encabezados), no suele figurar en línea de comando de *GCC*

2.10 Opciones.

`- c`

Realiza preprocesamiento y compilación, obteniendo el archivo en código objeto; no realiza el enlazado.

`- E`

Realiza solamente el preprocesamiento, enviando el resultado a la salida estándar.

`-o (archivo)`

Indica el nombre del archivo de salida, cualesquiera sean las etapas cumplidas.

`-I(ruta)`

Especifica la ruta hacia el directorio donde se encuentran los archivos marcados para incluir en el programa fuente. No lleva espacio entre la `I` y la ruta, así: `-I/usr/include`

`-L(ruta)`

Especifica la ruta hacia el directorio donde se encuentran los archivos de librerias con el código objeto de las funciones referenciadas en el programa fuente. No lleva espacio entre la `L` y la ruta, así: `-L/usr/lib`

`-Wall`

Muestra todos los mensajes de error y advertencia del compilador, incluso algunos cuestionables pero en definitiva fáciles de evitar escribiendo el código con cuidado.

`-g`

Incluye en el ejecutable generado la información necesaria para poder rastrear los errores usando un depurador, tal como GDB (*GNU Debugger*).

`-v`

Muestra los comandos ejecutados en cada etapa de compilación y la versión del compilador. Es un informe muy detallado.

2.11 Etapas de compilación.

El proceso de compilación involucra cuatro etapas sucesivas: preprocesamiento, compilación, ensamblado y enlazado. Para pasar de un programa fuente escrito por un humano a un archivo ejecutable es necesario realizar estas cuatro etapas en forma sucesiva. Los comandos `gcc` y `g++` son capaces de realizar todo el proceso de una sola vez.

2.12 Preprocesado.

En esta etapa se interpretan las directivas al preprocesador. Entre otras cosas, las variables inicializadas con `#define` son sustituidas en el código por su valor en todos los lugares donde aparece su nombre.

Se usa como ejemplo este programa de prueba llamado `circulo.c`:

```
/* Circulo.c: calcula el Área de un círculo.
   Ejemplo para mostrar etapas de compilación.
*/
#define PI 3.1416
main()
{
  float area, radio;
  radio = 10;
  area = PI * (radio * radio);
  printf("Circulo.\n");
  printf("%s%f\n\n", "Area de circulo radio 10: ", area);
}
```

El preprocesado puede solicitarse con cualquiera de los siguientes comandos; `cpp` alude específicamente al preprocesador.

```
$ gcc -E circulo.c > circulo.pp
$ cpp circulo.c > circulo.pp
```

Examinando `circulo.pp`

```
$ more circulo.pp
```

Se puede ver que la variable `PI` ha sido sustituida por su valor, 3.1416, tal como había sido fijado en la sentencia `#define`.

2.13 Compilación.

La compilación transforma el código *C* en el lenguaje ensamblador propio del procesador de la máquina donde se esta ejecutando.

```
$ gcc -S circulo.c
```

Realiza las dos primeras etapas creando el archivo `circulo.s`; examinándolo con

```
$ more circulo.s
```

Se puede ver el programa en lenguaje ensamblador.

2.14 Ensamblado.

El ensamblado transforma el programa escrito en lenguaje ensamblador a código objeto, un archivo binario en lenguaje de máquina ejecutable por el procesador. El ensamblador se denomina `as`:

```
$ as -o circulo.o circulo.s
```

Mediante este comando se crea el archivo en código objeto `circulo.o` a partir del archivo en lenguaje ensamblador `circulo.s`. No es frecuente realizar sólo el ensamblado; lo usual es realizar todas las etapas anteriores hasta obtener el código objeto así:

```
$ gcc -c circulo.c
```

Donde se crea el archivo `circulo.o` a partir de `circulo.c`. Puede verificarse el tipo de archivo usando el comando:

```
$ file circulo.o
circulo.o: ELF 32-bit LSB relocatable, Intel 80386, version 1, not stripped
```

En programas extensos, donde se escriben muchos archivos fuente en código *C*, es muy frecuente usar `gcc` o `g++` con la opción `-c` para compilar cada archivo fuente por separado, y luego enlazar todos los módulos objeto creados. Estas operaciones se automatizan colocándolas en un archivo llamado *Makefile*, interpretable por el comando `make`, quien se ocupa de realizar las actualizaciones mínimas necesarias toda vez que se modifica alguna porción de código en cualquiera de los archivos fuente.

2.15 Enlazado

Las funciones de *C/C++* incluidas en el código, tal como `printf()` en el ejemplo, se encuentran ya compiladas y ensambladas en bibliotecas existentes en el sistema. Es preciso incorporar de algún modo el código binario de estas funciones al archivo ejecutable. En esto consiste la etapa de enlace, donde se reúnen uno o más módulos en código objeto con el código existente en las bibliotecas.

El enlazador se denomina `ld`. Si se utiliza el siguiente comando para enlazar:

```
$ ld -o circulo circulo.o -lc
ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

Da este error por falta de referencias. Es necesario escribir el comando dándole todas las opciones para realizar el enlace y obtener un ejecutable.
:

```
  $ ld -o circulo /usr/lib/gcc-lib/i386-linux/2.95.2/collect2 -m elf_i386
-dynamic-linker    /lib/ld-linux.so.2    -o    circulo    /usr/lib/crt1.o
/usr/lib/crti.o      /usr/lib/gcc-lib/i386-linux/2.95.2/crtbegin.o      -
L/usr/lib/gcc-lib/i386-linux/2.95.2    circulo.o    -lgcc    -lc    -lgcc
/usr/lib/gcc-lib/i386-linux/2.95.2/crtend.o /usr/lib/crtn.o
```

El uso directo del enlazador `ld` es muy poco frecuente. En su lugar suele proveerse a `gcc` los códigos objeto directamente:

```
  $ gcc -o circulo circulo.o
```

Crea el ejecutable `circulo`, que invocado por su nombre ejecuta el programa `circulo`:

```
  $./circulo
  circulo
  Área de círculo radio 10: 314.160004
```

2.16 Todo en un solo paso.

En programas con un único archivo fuente todo el proceso anterior puede hacerse en un solo paso:

```
  $ gcc -o circulo circulo.c
```

No se crea el archivo `circulo.o`; el código objeto intermedio se crea y destruye sin verlo el operador, pero el programa ejecutable se crea y funciona.
En caso de problemas se puede usar la opción `-v` de `gcc` para obtener un informe detallado de todos los pasos de compilación:

```
$ gcc -v -o circulo circulo.c
```

3. DESCRIPCIÓN DE LIBRERÍAS PARA LA COMPILACIÓN DEL FIRMWARE

Para realizar la compilación de firmwares basados en openwrt es necesario contar con varias aplicaciones y librerías, estas se describen a continuación.

3.2 Binutils

Binutils es una colección de herramientas para el desarrollo de software que contiene un enlazador, un ensamblador y otras utilidades para trabajar con ficheros de objetos y archivos.

Programas instalados: addr2line, ar, as, c++filt, gprof, ld, nm, objcopy, objdump, ranlib, readelf, size, strings y strip

Librerías instaladas: libiberty.a, libbfd.[a,so] y libopcodes.[a,so]

3.2.1 Descripciones cortas

Addr2line*:* traslada direcciones de programas a nombres de ficheros y números de líneas. Dándole una dirección y un ejecutable, usa la información de depuración del ejecutable para averiguar qué archivo y número de línea está asociado con dicha dirección.

Arcrea: modifica y extrae desde archivos. Un archivo es un archivo que almacena una colección de otros ficheros en una estructura que hace posible obtener el original de cada archivo individual (llamados miembros del archivo).

As: es un ensamblador. Ensambla la salida de *GCC* dentro de ficheros objeto.

C++flit*:* es usado por el enlazador para decodificar (demangling) símbolos de *C++* y *Java*, guardando las funciones sobrecargadas para su clasificación.

Gprof*:* muestra los datos del perfil del gráfico de llamada.

Ld: es un enlazador. Combina un número de ficheros de objetos y de archivos en un único fichero, reubicando sus datos y estableciendo las referencias a los símbolos.

Nm: lista los símbolos que aparecen en un archivo objeto dado.

Objcopy*:* se utiliza para traducir un tipo de ficheros objeto a otro.

Objdump*:* muestra información sobre el archivo objeto indicado, con opciones para controlar la información a mostrar. La información mostrada es útil fundamentalmente para los programadores que trabajan sobre las herramientas de compilación.

Ranlib*:* genera un índice de los contenidos de un archivo, y lo coloca en el archivo. El índice lista cada símbolo definido por los miembros del archivo que son ficheros objeto reubicables.

Readelf: muestra información sobre binarios de tipo elf.

Size: lista los tamaños de las secciones y el tamaño total para cada uno de los ficheros objeto indicados.

Strings muestra, para cada archivo indicado, las cadenas de caracteres imprimibles de al menos la longitud especificada (4 por defecto). Para los ficheros objeto por defecto sólo muestra las cadenas procedentes de las secciones de inicialización y carga. Para otros tipos de ficheros explora el archivo al completo.

Strip: elimina símbolos de ficheros objeto.

Libiberty contiene rutinas usadas por varios programas GNU, incluidos getopt, obstack, strerror, strtol y strtoul.

Libbfd: es la librería del Descriptor de Archivo Binario.

Libopcodes es una librería para manejar mnemónicos. Se usa para construir utilidades como objdump. Los mnemónicos son las versiones en "texto legible" de las instrucciones del procesador.

3.2.2 Dependencias de instalación de Binutils

Binutils depende de: Bash, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Perl, Sed, Texinfo.

3.3 Patch

El programa patch modifica un archivo basándose en un parche. Normalmente un parche es una lista, creada por el programa diff, que contiene instrucciones sobre cómo debe modificarse el archivo original.

3.3.1 Descripción corta

Patch modifica ficheros según lo indicado en un archivo parche. Normalmente un parche es una lista de diferencias creada por el programa diff. Al aplicar estas diferencias a los ficheros originales, patch crea las versiones parcheadas. Usar parches en vez de un nuevo paquete completo para mantener actualizado el código fuente puede ahorrar un montón de tiempo de descarga.

3.3.2 Dependencias de instalación de Patch

Patch depende de: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Grep, Make, Sed.

3.4 Bzip2

Bzip2 es un compresor de ficheros por ordenación de bloques que, generalmente, consigue una mejor compresión que el tradicional gzip.

Programas instalados: bunzip2 (enlace a bzip2), bzcat (enlace a bzip2), bzcmp, bzdiff, bzegrep, bzfgrep, bzgrep, bzip2, bzip2recover, bzless y bzmore

Librerías instaladas: libbz2.a, libbz2.so *(*enlace a libbz2.so.1.0*)*, libbz2.so.1.0 *(enlace a* libbz2.so.1.0.2*)* y libbz2.so.1.0.2

3.4.1 Descripciones cortas

Bunzip2 descomprime ficheros que han sido comprimidos con bzip2.

Bzcat descomprime hacia la salida estándar.

Bzcmp ejecuta cmp sobre ficheros comprimidos con bzip2.

Bzdiff ejecuta diff sobre ficheros comprimidos con bzip2.

Bzgrep y sus derivados ejecutan grep sobre ficheros comprimidos con bzip2.

Bzip2 comprime ficheros usando el algoritmo de compresión de texto por ordenación de bloques Burrows-Wheeler con codificación Huffman. La compresión es, en general, considerablemente superior a la obtenida por otros compresores más convencionales basados en el LZ77/LZ78, como gzip.

Bzip2recover intenta recuperar datos de ficheros bzip2 dañados.

Bzless ejecuta less sobre ficheros comprimidos con bzip2.

Bzmore ejecuta more sobre ficheros comprimidos con bzip2.

Libbz2 es la librería que implementa la compresión sin pérdidas por ordenación de bloques, usando el algoritmo de Burrows-Wheeler.

3.4.2 Dependencias de instalación de Bzip2

Bzip2 depende de: Bash, Binutils, Coreutils, Diffutils, GCC, Glibc, Make.

3.5 Flex

El paquete Flex se utiliza para generar programas que reconocen patrones de texto.

Programas instalados: flex, flex++ *(*enlace a flex*) y* lex
Librería instalada: libfl.a

3.5.1 Descripciones cortas

Flex es una herramienta para generar programas capaces de reconocer patrones de texto. El reconocimiento de patrones es muy útil en muchas aplicaciones. A partir de un conjunto de reglas de búsqueda flex genera un programa que busca esos patrones. La razón para usar flex es porque es mucho más fácil establecer las reglas de búsqueda que escribir un programa real que busque el texto.

Flex++ invoca una versión de flex usada exclusivamente por analizadores C++.

Lbfl.a es la librería flex.

3.5.2 Dependencias de instalación de Flex

Flex depende de: Bash, Binutils, Bison, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, M4, Make, Sed.

3.6 Bison

Bison es un generador de analizadores sintácticos, un sustituto de yacc. Bison genera un programa que analiza la estructura de un archivo de texto.

Programas instalados: bison y yacc

Librería instalada: liby.a

3.6.1 Descripciones cortas

Bison genera, a partir de una serie de reglas, un programa para analizar la estructura de ficheros de texto. Bison es un sustituto de yacc (Yet Another Compiler Compiler, Otro Compilador de Compiladores).

Yacc es un envoltorio para bison, destinado a los programas que todavía llaman a yacc en lugar de a bison. Invoca a bison con la opción -y.

Liby.a es la librería Yacc que contiene la implementación de yyerror compatible con Yacc y funciones principales. Esta librería normalmente no es muy útil.

3.6.2 Dependencias de instalación de Bison

Bison depende de: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, M4, Make, Sed.

3.7 Make

Make determina, automáticamente, qué piezas de un programa largo es necesario recompilar y ejecuta los comandos para recompilarlas.

Programa instalado: make

3.7.1 Descripción corta

Make determina, automáticamente, qué partes de un paquete grande necesitan ser recompiladas y lanza los comandos para hacerlo.

3.7.2 Dependencias de instalación de Make

Make depende de: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Sed.

3.8 Gettext

El paquete Gettext se utiliza para la internacionalización y localización. Los programas pueden compilarse con soporte de lenguaje nativo (NLS, Native Language Support), lo que les permite mostrar mensajes en el idioma nativo del usuario.

Programas instalados: autopoint, config.charset, config.rpath, gettext, gettextize, hostname, msgattrib, msgcat, msgcmp, msgcomm, msgconv, msgen, msgexec, msgfilter, msgfmt, msggrep, msginit, msgmerge, msgunfmt, msguniq, ngettext, project-id, team-address, trigger, urlget, user-email y xgettext

Librerías instaladas: libasprintf[a,so], libgettextlib[a,so], libgettextpo[a,so] y libgettextsrc[a,so]

3.8.1 Descripciones cortas

Autopoint copìa los ficheros estándar de infraestructura de gettext a las fuentes de un paquete.

Config.charset saca una tabla dependiente del sistema de los alias de codificación de los caracteres.

Config.rpath saca un grupo de variables dependientes del sistema, describiendo cómo fijar la ruta de busqueda en tiempo de ejecución de las librerías compartidas en un ejecutable.

Gettext traduce un mensaje en lenguaje natural al lenguaje del usuario, buscando las traducciones en un catálogo de mensajes.

Gettextize copia todos los ficheros estándar *Gettext* en el directorio indicado de un paquete, para iniciar su internacionalización

`Hostname` muestra el nombre en la red de un sistema en varios formatos.

`Msgattrib` filtra los mensajes de un catálogo de traducción de acuerdo con sus atributos, y manipula dichos atributos.

`Msgcat` concatena y mezcla los ficheros `.p`*o* indicados.

`Msgcmp` compara dos ficheros `.po` para comprobar si ambos contienen el mismo conjunto de cadenas de identificadores de mensajes.

`Msgcomm` busca los mensajes comunes en los ficheros `.po` indicados.

`Msgconv` convierte un catálogo de traducción a una codificación de caracteres diferente.

`Mugen` crea un catálogo de traducción en inglés.

`Msgexec` aplica un comando a todas las traducciones de un catálogo de traducción.

`Msgfilter` aplica un filtro a todas las traducciones de un catálogo de traducción.

`Msgfmt` compila el binario de un catálogo de mensajes a partir de un catálogo de traducciones.

`Msggrep` extrae todos los mensajes de un catálogo de traducción que cumplan cierto criterio o pertenezcan a alguno de los ficheros fuentes indicadas.

`Msginit` crea un nuevo archivo `.po`, inicializando la información con valores procedentes del entorno del usuario.

`Msgmerge` combina dos traducciones directas en un único fichero.

`Msgunfmt` descompila catálogos de mensajes binarios en traducciones directas de texto.

`Msguniq` unifica las traducciones duplicadas en un catálogo de traducción.

`Ngettext` muestra traducciones en lenguaje nativo de un mensaje de textual cuya forma gramatical depende de un número.

`Xgettext` extrae las líneas de mensajes traducibles de los ficheros de los ficheros fuente indicados, para hacer la primera plantilla de traducción.

`Llibasprintf` defina la clase `autosprintf` que hace utilizable la salida formateada de las rutinas de *C* en programas C++, para usar con las cadenas <`string`> y los flujos <`iostream`>.

`Libgettextlib` es una librería privada que contiene rutinas comunes utilizadas por diversos programas de `gettext`. No está indicada para uso general.

`Libgettextpose` se usa para escribir programas especializados que procesan ficheros PO. Esta librería se utiliza cuando las aplicaciones estándar incluidas con `gettext` no son suficientes (como `msgcomm, msgcmp, msgattrib y msgen`).

`Libgettextsrc` es una librería privada que contiene rutinas comunes utilizadas por diversos programas de `gettext`. No está indicada para uso general.

3.8.2 Dependencias de instalación de Gettext

Gettext depende de: Bash, Binutils, Bison, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed.

3.9 Glibc

Glibc es la librería de *C* que proporciona las llamadas al sistema y las funciones básicas, tales como open, malloc, printf, etc. La librería *C* es utilizada por todos los programas enlazados dinámicamente.

Programas instalados: catchsegv, gencat, getconf, getent, glibcbug, iconv, iconvconfig, ldconfig, ldd, lddlibc4, locale, localedef, mtrace, nscd, nscd_nischeck, pcprofiledump, pt_chown, rpcgen, rpcinfo, sln, sprof, tzselect, xtrace, zdump y zic

Librerías instaladas: ld.so, libBrokenLocale.[a,so], libSegFault.so, libanl.[a,so], libbsd-compat.a, libc.[a,so], libc_nonshared.a, libcrypt.[a,so], libdl.[a,so], libg.a, libieee.a, libm.[a,so], libmcheck.a, libmemusage.so, libnsl.a, libnss_compat.so, libnss_dns.so, libnss_files.so, libnss_hesiod.so, libnss_nis.so, libnss_nisplus.so, libpcprofile.so, libpthread.[a,so], libresolv.[a,so], librpcsvc.a, librt.[a,so], libthread_db.so y libutil.[a,so]

3.9.1 Descripciones cortas

Catchsegv puede usarse para crear una traza de la pila cuando un programa termina con una violación de segmento.

Gencat genera catálogos de mensajes.

Getconf muestra los valores de configuración del sistema para variables especificas del sistema de ficheros.

Getent obtiene entradas de una base de datos administrativa.

Glibcbug crea un informe de fallos y lo envía a la dirección de correo electrónico de errores.

Iconv realiza conversiones de juego de caracteres.

Iconvconfig crea un archivo de configuración para la carga rápida del módulo *iconv*.

Ldconfig configura las asociaciones en tiempo de ejecución para el enlazador dinámico.

Ldd: muestra las librerías compartidas requeridas por cada programa o librería especificada.

lddlibc4 asiste a ld con los ficheros objeto.

`Locale` es un programa *Perl* que le dice al compilador si debe activar (o desactivar) el uso de las locales POSIX (Portable Operating System Interface) para operaciones integradas.

`Localedef` compila las especificaciones para locale.

`Nscd`: es un demonio que suministra una caché para las peticiones más comunes al servidor de nombres.

`Nscd_nischeck` comprueba si es necesario o no un modo seguro para búsquedas NIS+.

`Pcprofiledump` vuelca la información generada por un perfil de PC.

`Pt_chown` es un programa de ayuda para `grantpt` que establece el propietario, grupo y permisos de acceso para un seudo Terminal esclavo.

`Rpcgen` genera código *C* para implementar el protocolo RPC (Remote Procedure Calling).

`Rpcinfo` hace una llamada RPC en un servidor RPC.

`Sln` se usa para hacer enlaces simbólicos. Está enlazado estáticamente, por lo que es útil para crear enlaces simbólicos a librerías dinámicas si, por alguna razón, el enlazador dinámico del sistema no funciona.

`Sprof` lee y muestra los datos del perfil de los objetos compartidos.

`Tzselect` pregunta al usuario información sobre la localización actual y muestra la descripción de la zona horaria correspondiente.

`Xtrace` traza la ejecución de un programa mostrando la función actualmente ejecutada.

`Zdump` es el visualizador de la zona horaria.

`Zic` es el compilador de la zona horaria.

`Ld.so` es el programa de ayuda para las librerías compartidas ejecutables.

`LibBrokenLocale` es usada por programas como Mozilla para resolver locales rotas.

`LibSegFault` es un manejador de señales de violación de segmento. Intenta capturar estas señales.

`Libanl` es una librería de búsqueda de nombres asíncrona.

`Libbsd-compat` proporciona la portabilidad necesaria para ejecutar ciertos programas BSD en Linux.

`Libc` es la librería principal de C, una colección de funciones usadas frecuentemente.

`Libcrypt` es la librería criptográfica.

`Libdl` es la librería de interfaz del enlazado dinámico.

`Libg` es una librería en tiempo de ejecución de `g++`.

`Libieee` es la librería de punto flotante IEEE.

`Libm` es la librería matemática.

`Libmcheck` contiene código ejecutado en el arranque.

`Libmemusage` es usada por `memusage` para ayudar a recoger información sobre el uso de memoria de un programa.

`Libnsl` es la librería de servicios de red.

`Libnss` son las librerías del Interruptor del Servicio de Nombres (*NSS, Name Service Switch*). Contienen funciones para resolver el nombre de sistemas, de usuarios, de grupos, alias, servicios, protocolos y similares.

`Libpcprofile` Código usado por el núcleo para rastrear el tiempo de CPU gastado en funciones, líneas de código fuente e instrucciones.

`Libpthread` es la librería de hilos POSIX.

`Libresolv` proporciona funciones para la creación, envío e interpretación de paquetes de datos a servidores de nombres de dominio de Internet.

`Librpcsvc` proporciona funciones para una miscelánea de servicios RPC.

`Librt` proporciona funciones para muchas de las interfaces especificadas por el POSIX.1b Realtime Extension (Extensiones en Tiempo Real POSIX.1b).

`Libthread_db` contiene funciones útiles para construir depuradores para programas multihilo.

`Libutil` contiene código para funciones "estándar" usadas en diferentes utilidades *Unix*.

3.9.2 Dependencias de instalación de `Glibc`

`Glibc` depende de: `Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Gettext, Grep, Make, Perl, Sed, Texinfo.`

4 ENLACE DINÁMICO Y ESTÁTICO DE LIBREIRAS

Existen dos modos de realizar el enlace:

4.1 Enlace Estático

Los binarios de las funciones se incorporan al código binario del ejecutable, de manera que no se necesitan las librerías después de generado el ejecutable.

4.2 Enlace Dinámico

El código de las funciones de un programa enlazado dinámicamente permanece en la librería del sistema, el ejecutable carga en memoria la biblioteca y ejecuta la parte de código correspondiente en el momento de correr el programa.
El enlazado dinámico permite crear un ejecutable más pequeño, pero requiere el acceso a las bibliotecas en el momento de correr el programa. El enlazado estático crea un programa autónomo, pero al precio de agrandar el tamaño del ejecutable binario.

Ejemplo de enlazado estático:

```
$ gcc -static -o circulo circulo.c
```

Ahora se observa el tamaño del archivo mediante el comando `ls`

```
$ ls -l circulo
-rwxr-xr-x    1 lxuser   lxuser     237321 ago  4 11:24 circulo
```

Si no se especifica `-static` el enlazado es dinámico por defecto.

Ejemplo de enlazado dinámico:

```
$ gcc -o circulo circulo.c
```

Ahora se observa el tamaño del archivo mediante el comando `ls`

```
$ ls -l circulo
-rwxr-xr-x    1 lxuser   lxuser       4828 ago  4 11:26 circulo
```

Se puede notar la diferencia en tamaño del ejecutable compilado estática o dinámicamente. Los valores pueden diferir en algo de los mostrados; dependen de la plataforma y la versión del compilador.

El comando `ldd` muestra las dependencias de bibliotecas compartidas que tiene un ejecutable:

```
$ gcc -o circulo circulo.c
$ ldd circulo
        libc.so.6 => /lib/libc.so.6 (0x40017000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

```
$ gcc -static -o circulo circulo.c
$ ldd circulo
        statically linked (ELF)
```

La compilación estática no muestra ninguna dependencia de biblioteca.

5 UTILIDAD MAKE

El comando de Linux `make` es una ayuda para compilar programas. Presenta ventajas para los programas grandes en los que hay varios archivos fuente (`.c` y.`h`) repartidos por varios directorios. Principalmente aporta dos ventajas:

- Es capaz de saber qué archivos hay que recompilar. Si se esta depurando un programa y se modifica un archivo fuente, al compilar con `make` sólo se recompilaran aquellos archivos que dependan de los que se han modificado. Si se compila a mano con `gcc`, (u otro compilador), se debe tener en mente las dependencias para compilar sólo lo que hace falta, o compilar todo nuevamente. Si el proyecto es grande, es posible que se pase por alto una dependencia o que el tiempo de compilación sea muy largo.

- `Make` guarda los comandos de compilación con todos sus parámetros para encontrar librerías, archivos de cabecera (`.h`), etc. Sólo se es necesario escribir las líneas de compilación con las opciones una sola vez dentro del `makefile`.

A continuación se muestra un ejemplo básico para mostrar el funcionamiento de `make`

```
#include <stdio.h>
main()
{
    printf ("Hola Mundo\n");
}
```

Se compila de la forma habitual.

```
$ cc HolaMundo.c -o HolaMundo
```

Se prueba de la siguiente manera:

```
$ ./HolaMundo
```

Y el resultado es el siguiente:

```
$ Hola Mundo
```

Este resultado muestra que se ejecuta correctamente el programa, ahora se compila con `make`.

```
$ make HolaMundo
```

`Make` dice que no hay nada que hacer, esta es una diferencia con compilar a mano, debido a que el programa ya está hecho, `make` no hace nada. Esto, en un programa de cientos de líneas de código que tarda varios minutos en compilar, es una gran ventaja.

Ahora se borra el ejecutable se ejecuta nuevamente `make`

```
$ rm HolaMundo
$
$ make HolaMundo
$
```

Esta vez si se compila el `Hola Mundo`.

Esto quiere decir que `make` revisa si el ejecutable ya está hecho y compara además las fechas de la fuente (el `HolaMundo.c`) y el ejecutable, compilando o no el programa en función de estas fechas.

`Make` entiende además extensiones y compiladores. Al hacer `make HolaMundo`, el archivo `HolaMundo.c` se compila con el compilador `gcc`, pero si se cambia la extensión a `HolaMundo.cpp`, al ejecutar `make` se compilará con el compilador de `c++`, el `g++`. También entiende extensiones de *fortran* y de ensamblador. Las versiones más modernas (como *Solaris*, el *Unix* de Sun) incluso son capaces de compilar *Java*.

Todo esto se debe a que `make` tiene unas reglas implícitas que le indican cómo obtener un ejecutable. Estas reglas en *Solaris*, están escritas en un archivo `make.rules`. En Linux generalmente se encuentra en `/usr/share/lib/make/make.rules`.

5.1 Variables de entorno

Como un segundo ejemplo se crean dos directorios, uno de ellos se llama `PRINCIPAL` y el otro `FUNCION1`. En `PRINCIPAL` se guarda el archivo `HolaMundo.c`. En `FUNCION1` se crea un archivo `.h` con nombre `texto.h` y su contenido es este:

```
#define TEXTO "Hola Mundo"
```

En el archivo `HolaMundo.c` se coloca lo siguiente:

```
#include <stdio.h>
#include <texto.h>
main()
{
    printf ("%s\n", TEXTO);
}
```

Para compilar este archivo se debe hacer uso de la variable de entorno `CFLAGS` para el compilador de *C* o `CPPFLAGS` para el compilador C++, la cual puede contener las opciones que se pasen al compilador. Una de las opciones que se puede pasar al compilador es la opción `-I`, cual permite establecer `paths` o rutas de búsqueda para archivos de cabecera *(`.h`).* En este ejemplo, y usando un `path` relativo, el comando se aplica de la siguiente manera:

```
$ CFLAGS=-I../FUNCION1; export CFLAGS
$ make HolaMundo
$
```

Esta vez se compila el programa.

5.2 El archivo `Makefile`

En el momento que se tienen dos o mas archivos `.c` para construir un único ejecutable, `make` necesita saber cuáles son los archivos que debe compilar, para esto se puede hacer que `make`

ejecute un archivo, uno de cuyos nombres por defecto es `Makefile`, y establecer en él qué comandos se van a ejecutar y cómo.
Este `makefile` tiene las siguientes partes:

```
objetivo: dependencia1 dependencia2 ...
<tab>comando1
<tab>comando2
<tab>...
```

- Objetivo: Es lo que se quiere construir, este puede ser el nombre de un ejecutable, el nombre de una librería o cualquier palabra. Para este ejemplo, el objetivo es el nombre del ejecutable, es decir `HolaMundo`.

- dependencia<i>: Es el nombre de un objetivo del cual depende el objetivo anterior, es necesario que las dependencias se ejecuten antes que el objetivo. En el ejemplo, las dependencias son los archivos fuente `HolaMundo.c`, `../FUNCION1/FUNCION1.h` y `../FUNCION1/FUNCION1.c` ya que para hacer el ejecutable se necesitan todas esas fuentes.

- <tab>: Es un tabulador. Es importante para que el archivo sea leído correctamente.

- comando<i>: Es lo que se debe ejecutar para construir el objetivo, estos comandos se van ejecutando en secuencia. Puede ser cualquier comando de shell válido. (`cc`, `rm`, `cp`, `ls`, o incluso un *script*). En el ejemplo, esl comando es:
  `cc HolaMundo.c -o HolaMundo`.

De esta manera el `makefile` solo necesita el siguiente par de líneas:

```
HolaMundo: HolaMundo.c ../FUNCION1/FUNCION1.c ../FUNCION1/FUNCION1.h
      cc -I../FUNCION1 HolaMundo.c ../FUNCION1/FUNCION1.c -o HolaMundo
```

En este momento se puede ejecutar `make`, (sin parámetro) y se vuelve a compilar el programa. Si a `make` no se le introduce algún parámetro, busca un archivo `Makefile` y dentro de él realiza el primer objetivo que encuentre. En este caso, el único que hay es `HolaMundo`.
Se puede observar el comando de compilación, el `cc -I`, se ha colocado la opción `-I../FUNCION1` para `make` que sea capaz de encontrar los archivos `.h`. Si se coloca un comando de compilación, ya no valen las reglas implícitas que conoce `make`. Al colocar un comando de compilación, el ejecutable se generará con el comando (regla explícita) y no utiliza nada de otro sitio (ni siquiera la variable de entorno `CFLAGS`).

5.3 Mejora del `Makefile`

Este `Makefile` funciona correctamente, si se modifica o cambia la fecha de creación del archivo (con `touch` o editándolo y salvando), se recompilará el programa, pero para aprovechar la ventaja que ofrece `make` para proyectos grandes, la cual consiste en compilar sólo aquellas partes que son necesarias, se deben hacer unos ajustes. Lo ideal, es que si se modifica `FUNCION1.h`, sólo

se recompile HolaMundo.c, que incluye FUNCION1.h, pero que no se recompile FUNCION1.c, que no incluye a FUNCION1.h.

Para conseguir este comportamiento se debe tener guardados los archivos objeto (los .o) que se generan. En un proyecto grande, estos .o se pueden guardar en forma de librerías. Para este ejemplo lo se dejan los .o.

El ejecutable HolaMundo depende únicamente de los dos .o correspondientes a los .c.

En este caso el Makefile queda de la siguiente manera:

```
HolaMundo: HolaMundo.o ../FUNCION1/FUNCION1.o
	cc HolaMundo.o ../FUNCION1/FUNCION1.o -o HolaMundo
```

Con esto ya no se necesita la opción -I../FUNCION1 ya que los archivos fuente ya están compilados y no necesitan los .h.

Aunque make puede hacer los .o de una forma por defecto, (busca el .c con el mismo nombre y lo compila con las opciones adecuadas), sigue sin saber encontrar los .h que estén en otros directorios, pero es posible colocar más reglas en el Makefile para que sepa hacer correctamente los .o. Estas se colocan después, ya que make hace el primer objetivo que encuentre en el archivo y el primer objetivo debe ser el ejecutable.
Ahora el Makefile queda de la siguiente manera:

```
HolaMundo: HolaMundo.o ../FUNCION1/FUNCION1.o
	cc HolaMundo.o ../FUNCION1/FUNCION1.o -o HolaMundo

../FUNCION1/FUNCION1.o: ../FUNCION1/FUNCION1.c
	cc -c ../FUNCION1/FUNCION1.c -o ../FUNCION1/FUNCION1.o

HolaMundo.o: HolaMundo.c ../FUNCION1/FUNCION1.h
	cc -c -I../FUNCION1 HolaMundo.c -o HolaMundo.o
```

En este momento, cuando se ejecute make, se tratará de hacer el primer objetivo, HolaMundo. Como este depende de dos .o, se harán primero estos dos. Cada uno de ellos depende de los fuentes (.c y .h) correspondientes. Si las fuentes son más modernas que el .o o si no existe el .o, se compilará. Si el .o es más moderno que las fuentes, no se hará nada. Una vez construidos los .o, se construirá HolaMundo en función de que las fechas de estos .o sean más modernas o no que el ejecutable HolaMundo.

Para la primera compilación se realizan todos los objetivos ya que ninguno se encuentra construido. Una vez realizada esta primera compilación se puede verificar el funcionamiento de make editando y salvando el archivo FUNCION1.h y luego ejecutando make, el comando realiza los siguientes pasos:

- HolaMundo depende de HolaMundo.o y ../FUNCION1/FUNCION1.o. se verifican estos dos antes de hacer nada.

- HolaMundo.o depende de HolaMundo.c y ../FUNCION1/FUNCION1.h. Como se ha modificado FUNCION1.h, este es más moderno que HolaMundo.o, así que se hace la compilación.
- ../FUNCION1/FUNCION1.o depende de ../FUNCION1/funcion.c. Como no se ha modificado ninguno de estos, el .o será más moderno que el .c y no se compila nada.
- Como HolaMundo.o se ha generado nuevo, es más moderno que el ejecutable HolaMundo, así que se compila.

Con esto se observa que sólo se ha compilado uno de los .c, mientras que el otro no se ha compilado.

Hay que recordar que make tiene unas reglas implícitas que le permiten construir los .o, make sólo necesita saber dónde encontrar los archivos de cabecera (.h). Además, para un .o únicamente hace falta un .c, así que basta con la regla implícita y la variable de entorno CFLAGS. El Makefile queda de la siguiente manera:

```
HolaMundo: HolaMundo.o ../FUNCION1/FUNCION1.o
    cc HolaMundo.o ../FUNCION1/FUNCION1.o -o HolaMundo

../FUNCION1/FUNCION1.o: ../FUNCION1/FUNCION1.c

HolaMundo.o: HolaMundo.c ../FUNCION1/FUNCION1.h
```

Para que funcione correctamente, antes de hacer make, se define la variable de entorno

```
$ CFLAGS=-I../FUNCION1; export CFLAGS
$ make
```

En el Makefile solo se colocan las dependencias de los .o con los archivos fuente. Esto es necesario al estar las fuentes por varios directorios y tener archivos .h. La regla implícita para los .o únicamente hará depender al .o de un .c con el mismo nombre y que esté en el mismo directorio.

5.4 Variables en el Makefile

Es posible definir las variables de entorno en el makefile, de manera que no se necesite definirlas mediante un comando, además se pueden definir en una variable los .o, de HolaMundo para no sea necesario escribirlos dos veces (una en las dependencias de HolaMundo y otra en el comando de compilación). El archivo Makefile, después de estos cambios, queda de la siguiente manera:

```
OBJETOS=HolaMundo.o ../FUNCION1/FUNCION1.o
CFLAGS=-I../FUNCION1

HolaMundo: $(OBJETOS)
    cc $(OBJETOS) -o HolaMundo
```

```
../FUNCION1/FUNCION1.o: ../FUNCION1/FUNCION1.c

HolaMundo.o: HolaMundo.c ../FUNCION1/FUNCION1.h
```

Para asignar una variable se coloca el nombre de la variable, un igual y lo que su contenido, si la línea es muy larga, se puede partir con la el caracter \

```
OBJETOS=HolaMundo.o \
    ../FUNCION1/FUNCION1.o
```

Hay que asegurarse que inmediatamente detrás de la `\` está el retorno de carro, que no haya ningún espacio ni nada parecido detrás del caracter `\` .
Para usar el contenido de la variable, se coloca `$` y entre paréntesis el nombre de la variable.

5.5 Varios `makefiles`

En un proyecto grande, los `.o` se suelen guardar en librerías. Lo habitual es que en cada directorio de fuentes se haga un `Makefile` específico para construir una librería con las fuentes de ese directorio. El `Makefile` principal llama a los `Makefile` de cada directorio (porque ese será el comando de compilación que se coloque en el principal) y luego construya el primer `makefile`.
Es decir, si con `FUNCION1.c` se hace la librería `libFUNCION1.a` y tiene su propio `Makefile` en su propio directorio `FUNCION1`, el `Makefile` principal queda de la siguiente manera:

```
CFLAGS=-I../FUNCION1
LDFLAGS=-L../FUNCION1
HolaMundo : HolaMundo.o ../FUNCION1/libFUNCION1.a
      cc -o HolaMundo HolaMundo.o $(LDFLAGS) -lFUNCION1
../FUNCION1/libFUNCION1.a:
      make -C ../FUNCION1
```

`LDFLAGS` es una variable igual que `CFLAGS`, pero que le dice a las reglas implícitas qué opciones usar para el *linkado*. En este caso se usa la opción `-L`, que dice dónde hay librerías. No hay ninguna regla implícita en el `Makefile` que lo vaya a usar, pero se llama a la variable de la misma manera.

Para hacer la librería, se llama a `make` con la opción `-C`. Esta opción le dice a `make` que debe cambiarse al directorio `../FUNCION1` antes de ejecutarse. Allí encontrará el `Makefile` para hacer la librería. En el `Makefile` de `PRINCIPAL` no se colocan las dependencias para la librería. Estas dependencias se encuentran en el `Makefile` de `FUNCION1` y será este el que decida si hay que reconstruir o no la librería.